The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus

Beniamino Accattoli

The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus

Beniamino Accattoli

INRIA & LIX, École Polytechnique
beniamino.accattoli@inria.fr

Abstract. It has been a long-standing open problem whether the strong λ-calculus is a reasonable computational model, i.e. whether it can be implemented within a polynomial overhead with respect to the number of β-steps on models like Turing machines or RAM. Recently, Accattoli and Dal Lago solved the problem by means of a new form of sharing, called *useful sharing*, and realised via a calculus with explicit substitutions. This paper presents a new abstract machine for the strong λ-calculus based on useful sharing, the *Useful Milner Abstract Machine*, and proves that it reasonably implements leftmost-outermost evaluation. It provides both an alternative proof that the λ-calculus is reasonable and an improvement on the technology for implementing strong evaluation.

1 Introduction

The higher-order computational model of reference is the λ-calculus, that comes in two variants, *weak* or *strong*. Introduced at the inception of computer science as a mathematical approach to computation, it later found applications in the theoretical modelling of programming languages and, more recently, proof assistants. The weak λ-calculus is the backbone of functional languages such as LISP, Scheme, OCAML, or Haskell. It is *weak* because evaluation does not enter function bodies and, usually, terms are assumed to be closed. By removing these restrictions one obtains the *strong* λ-calculus, that underlies proof assistants like Coq, Isabelle, and Twelf, or higher-order logic programming languages such as λ-prolog or the Edinburgh Logical Framework. Higher-order features nowadays are also part of mainstream programming languages like Java or Python.

The abstract, mathematical character is both the advantage and the drawback of the higher-order approach. The advantage is that it enhances the modularity and the conciseness of the code, allowing to forget about low-level details at the same time. The drawback is that the distance from low-level details makes its complexity harder to analyse, in particular its main computational rule, called β-*reduction*, at first sight is not an atomic operation. In particular, β can be nasty, and make the program grow at an exponential rate. The number of β-steps, then, does not even account for the time to write down the result, suggesting that it is not a reasonable cost model. This is the *size-explosion problem* [6], and affects both the weak and the strong λ-calculus.

The λ-Calculus is Reasonable, Indeed A cornerstone of the theory is that, nonetheless, in the weak λ-calculus the number of β-steps *is* a reasonable cost model for time complexity analyses [9,25,14], where *reasonable* formally means that it is polynomially related to the cost model of RAM or Turing machines.

For the strong λ-calculus, the techniques developed for the weak one do not work, as wilder forms of size-explosion are possible. A natural candidate cost model from the theory of λ-calculus is the number of (Lévy) optimal parallel steps, but it has been shown by Asperti and Mairson that such a cost model is not reasonable [8].

It is only very recently that the strong case has been solved by Accattoli and Dal Lago [6], who showed that the number of leftmost-outermost β-steps to full normal form is a reasonable cost model. The proof of this result relies on two theoretical tools. First, the *Linear Substitution Calculus* (LSC), an expressive and simple decomposition of the λ-calculus via linear logic and rewriting theory, developed by Accattoli and Kesner [3] as a variation over a calculus by Robin Milner [24]. Second, *useful sharing*, a new form of shared evaluation introduced by Accattoli and Dal Lago on top of the LSC. Roughly, the LSC is a calculus where the meta-level operation of substitution used by β-reduction is internalised and decomposed in micro steps, *i.e.* it is what is usually called a calculus with *explicit substitutions*. The further step is to realise that some of these micro substitution steps are useless: they do not lead to the creation of other β-redexes, their only aim is to unshare the result and provide the full normal form. Useful evaluation then performs only those substitution steps that are useful, *i.e.* not useless. By avoiding useless unsharing steps, it computes a shared representation of the normal form of size linear in the number of steps, whose unsharing may cause an exponential blow up in size. This is how the size-explosion problem is circumvented, see [6] for more explanations.

This Paper In this paper we provide an alternative proof that the strong λ-calculus is reasonable (actually only of the hard half, that is the simulation of λ-calculus on RAM, the other half being much easier, see [5]), by replacing the LSC with the *Useful Milner Abstract Machine*. The aim of the paper is threefold:

1. *Getting Closer To Implementations*: the LSC decomposes β-reduction in micro-steps but omits details about the search for the next redex to reduce. Moreover, in [6] useful sharing is used as a sort of black box on top of the LSC. Switching to abstract machines provides a solution closer to implementations and internalises useful sharing.
2. *The First Reasonable Strong Abstract Machine*: the literature on abstract machines for strong evaluation is scarce (see below) and none of the machines in the literature is reasonable. This work thus provides an improvement of the technology for implementing strong evaluation.
3. *Alternative Proof*: the technical development in [6] is sophisticated, because a second aim of that paper is to connect some of the used tools (namely useful sharing and the subterm property) with the seemingly unrelated notion of *standardisation* from rewriting theory. Here we provide a more basic, down-to-earth approach, not relying on advanced rewriting theory.

The Useful MAM The Milner Abstract Machine (MAM) is a variant with just one *global environment* of the Krivine Abstract Machine (KAM), introduced in [1] by Accattoli, Barenbaum, and Mazza. The same authors introduce in [2] the Strong MAM, *i.e.* the extension of the MAM to strong evaluation, that is a version with just one global environment of Cregut's Strong KAM [13], essentially the only other abstract machine for strong (call-by-name) evaluation in the literature. Both are not reasonable. The problem is that these machines do not distinguish between useful and useless steps.

The Useful MAM introduced in this paper improves the situation, by refining the Strong MAM. The principle is quite basic, let us sketch it. Whenever a β-redex $(\lambda x.t)u$ is encountered, the Strong MAM adds an entry $[x{\leftarrow}u]$ to the environment E. The Useful MAM, additionally, executes an auxiliary machine on u—the Checking Abstract Machine (Checking AM)—to establish its usefulness. The result of this check is a label l that is attached to the entry $[x{\leftarrow}u]^l$. Later on, when an occurrence of x is found, the Useful MAM replaces x with u only if the label on $[x{\leftarrow}u]^l$ says that it is useful. Otherwise the machine backtracks, to search for the next redex to reduce.

The two results of the paper are:

1. *Qualitative* (Theorem 2): the Useful MAM correctly and completely implements leftmost-outermost (LO for short) β-evaluation—formally, the two are weakly bisimilar.
2. *Quantitative* (Theorem 5): the Useful MAM is a reasonable implementation, *i.e.* the work done by both the Useful MAM and the Checking AM is polynomial in the number of LO β-steps and in the size of the initial term.

Related Work Beyond Crégut's [12,13] and Accattoli, Barenbaum, and Mazza's [2], we are aware of only two other works on strong abstract machines, García-Pérez, Nogueira and Moreno-Navarro's [22] (2013), and Smith's [27] (unpublished, 2014). Two further studies, de Carvalho's [11] and Ehrhard and Regnier's [19], introduce strong versions of the KAM but for theoretical purposes; in particular, their design choices are not tuned towards implementations (*e.g.* rely on a naïve parallel exploration of the term). Semi-strong machines for call-by-value (*i.e.* dealing with weak evaluation but on open terms) are studied by Grégoire and Leroy [23] and in a recent work by Accattoli and Sacerdoti Coen [4] (see [4] for a comparison with [23]). More recent work by Dénès [18] and Boutiller [10] appeared in the context of term evaluation in Coq. None of the machines for strong evaluation in the literature is reasonable, in the sense of being polynomial in the number of β-steps. The machines developed by Accattoli and Sacerdoti Coen in [4] are reasonable, but they are developed in a semi-strong setting only. Another difference between [4] and this work is that call-by-value simplifies the treatment of usefulness because it allows to compute the labels for usefulness while evaluating the term, that is not possible in call-by-name.

Global environments are explored by Fernández and Siafakas in [20], and used in a minority of works, *e.g.* [25,17]. Here we use the terminology for abstract machines coming from the distillation technique in [1], related to the *refocusing* *semantics* of Danvy and Nielsen [16] and introduced to revisit the relationship between the KAM and weak linear head reduction pointed out by Danos and Regnier [15]. We do not, however, employ the distillation technique itself.

Proofs All proofs have been omitted. Those of the main lemmas and theorems concerning the Useful MAM can be found in the appendix. The other ones can be found in the longer version on the author's web page.

2 λ-Calculus and Leftmost-Outermost Evaluation

The syntax of the λ-calculus is given by the following grammar for terms:

$$\lambda\text{-Terms} \qquad t, u, w, r ::= x \mid \lambda x.t \mid tu.$$

We use $t\{x{\leftarrow}u\}$ for the usual (meta-level) notion of substitution. An abstraction $\lambda x.t$ binds x in t, and we silently work modulo α-equivalence of bound variables, *e.g.* $(\lambda y.(xy))\{x{\leftarrow}y\} = \lambda z.(yz)$. We use $\mathtt{fv}(t)$ for the set of free variables of t.

Contexts. One-hole contexts C and the *plugging* $C\langle t\rangle$ of a term t into a context C are defined by:

$$\begin{array}{c} \text{Contexts} \\ C ::= \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC \end{array} \qquad \begin{array}{ccc} & \text{Plugging} & \\ \langle\cdot\rangle\langle t\rangle \;:=\; t & & (Cu)\langle t\rangle := C\langle t\rangle u \\ (\lambda x.C)\langle t\rangle := \lambda x.C\langle t\rangle & & (uC)\langle t\rangle := uC\langle t\rangle \end{array}$$

As usual, plugging in a context can capture variables, *e.g.* $(\lambda y.(\langle\cdot\rangle y))\langle y\rangle = \lambda y.(yy)$. The plugging $C\langle C'\rangle$ of a context C' into a context C is defined analogously. A context C is *applicative* if $C = C'\langle\langle\cdot\rangle\overline{u}\rangle$ for some C' and $\overline{u}$.

We define β-reduction $\rightarrow_\beta$ as follows:

$$\begin{array}{c} \text{Rule at Top Level} \\ (\lambda x.t)u \mapsto_\beta t\{x{\leftarrow}u\} \end{array} \qquad \begin{array}{c} \text{Contextual closure} \\ C\langle t\rangle \rightarrow_\beta C\langle u\rangle \quad \text{if } t \mapsto_\beta u \end{array}$$

A term t is a *normal form*, or simply *normal*, if there is no u such that $t \rightarrow_\beta u$, and it is *neutral* if it is normal and it is not of the form $\lambda x.u$ (*i.e.* it is not an abstraction). The *position* of a β-redex $C\langle t\rangle \rightarrow_\beta C\langle u\rangle$ is the context C in which it takes place. To ease the language, we will identify a redex with its position. A *derivation* $d : t \rightarrow^k u$ is a finite, possibly empty, sequence of reduction steps. We write $|t|$ for the size of t and $|d|$ for the length of d.

Leftmost-Outermost Derivations The left-to-right outside-in order on redexes is expressed as an order on positions, *i.e.* contexts.

Definition 1 (Left-to-Right Outside-In Order).

1. *The* outside-in order*:*
   (a) Root*:* $\langle\cdot\rangle \prec_O C$ *for every context* $C \neq \langle\cdot\rangle$*;*
   (b) Contextual closure*: If* $C \prec_O C'$ *then* $C''\langle C\rangle \prec_O C''\langle C'\rangle$ *for any* C''*.*
2. *The* left-to-right order*:* $C \prec_L C'$ *is defined by:*

   (a) Application*: If $C \prec_p t$ and $C' \prec_p u$ then $Cu \prec_L tC'$;*
   (b) Contextual closure*: If $C \prec_L C'$ then $C''\langle C\rangle \prec_L C''\langle C'\rangle$ for any C''.*
3. *The* left-to-right outside-in order*: $C \prec_{LO} C'$ if $C \prec_O C'$ or $C \prec_L C'$:*

The following are a few examples. For every context C, it holds that $\langle\cdot\rangle \not\prec_L C$. Moreover $(\lambda x.\langle\cdot\rangle)t \prec_O (\lambda x.(\langle\cdot\rangle u))t$ and $(\langle\cdot\rangle t)u \prec_L (wt)\langle\cdot\rangle$.

Definition 2 (LO β-Reduction). *Let t be a λ-term and C a redex of t. C is the* leftmost-outermost *β-redex (LO β for short) of t if $C \prec_{LO} C'$ for every other β-redex C' of t. We write $t \to_{\mathtt{LO}\beta} u$ if a step reduces the LO β-redex.*

The next immediate lemma guarantees that we defined a total order.

Lemma 1 (Totality of $\prec_{LO}$). *If $C \prec_p t$ and $C' \prec_p t$ then either $C \prec_{LO} C'$ or $C' \prec_{LO} C$ or $C = C'$. Therefore, $\to_{\mathtt{LO}\beta}$ is deterministic.*

LO Contexts For the technical development of the paper we need two characterisations of when a context is the position of the LO β-redex. The first, following one, is used in the proofs of Lemma 5.2 and Lemma 6.4.

Definition 3 (LO Contexts). *A context C is* LO *if*

1. Right Application*: whenever $C = C'\langle tC''\rangle$ then t is neutral, and*
2. Left Application*: whenever $C = C'\langle C''t\rangle$ then $C'' \neq \lambda x.C'''$.*

The second characterisation is inductive, and it used to prove Lemma 10.3.

Definition 4 (iLO Context). *Inductive LO β (or iLO) contexts are defined by induction as follows:*

$$\frac{}{\langle\cdot\rangle \text{ is iLO}}\ (\text{ax-iLO}) \qquad \frac{C \text{ is iLO} \quad C \neq \lambda x.C'}{Ct \text{ is iLO}}\ (\text{@l-iLO})$$

$$\frac{C \text{ is iLO}}{\lambda x.C \text{ is iLO}}\ (\lambda\text{-iLO}) \qquad \frac{t \text{ is neutral} \quad C \text{ is iLO}}{tC \text{ is iLO}}\ (\text{@r-iLO})$$

As expected,

Lemma 2 ($\to_{\mathtt{LO}\beta}$-steps and Contexts). *Let t be a λ-term and C a redex in t. C is the LO β redex in t iff C is LO iff C is iLO.*

3 Preliminaries on Abstract Machines

We study two abstract machines, the Useful MAM (Fig. 4) and an auxiliary machine called the Checking AM (Fig. 2).

The Useful MAM is meant to implement LO β-reduction strategy via a decoding function $\underline{\cdot}$ mapping machine states to λ-terms. Machine states s are given by a *code* $\overline{t}$, that is a λ-term t *not considered up to α-equivalence* (which is why it is over-lined), and some data-structures like stacks, frames, and environments. The data-structures are used to implement the search for the next *LO*-redex and a form of micro-steps substitution, and they decode to evaluation contexts for $\rightarrow_{\mathtt{LO}\beta}$. Every state s decodes to a term $\underline{s}$, having the shape $C_s\langle\overline{t}\rangle$, where $\overline{t}$ is the code currently under evaluation and C_s is the evaluation context given by the data-structures.

The Checking AM tests the usefulness of a term (with respect to a given environment) and outputs a label with the result of the test. It uses the same states and data-structures of the Useful MAM.

The Data-Structures First of all, our machines are executed on *well-named* terms, that are those α-representants where all variables (both bound and free) have distinct names. Then, the data-structures used by the machines are defined in Fig. 1, namely:

- *Stack* π: it contains the arguments of the current code;
- *Frame* F: a second stack, that together with π is used to walk through the term and search for the next redex to reduce. The items ϕ of a frame are of two kinds. A variable x is pushed on the frame F whenever the machines starts evaluating under an abstraction λx. A *head argument context* $\overline{t}\Diamond\pi$ is pushed every time evaluation enters in the right subterm $\overline{u}$ of an application $\overline{t}\overline{u}$. The entry saves the left part $\overline{t}$ of the application and the current stack π, to restore them when the evaluation of the right subterm $\overline{u}$ is over.
- *Global Environment* E: it is used to implement micro-step evaluation (*i.e.* the substitution on a variable occurrence at the time), storing the arguments of β-redexes that have been encountered so far. Most of the literature on abstract machines uses *local environments* and *closures*. Having just one global environment E removes the need for closures and simplifies the machine. On the other hand, it forces to use explicit α-renamings (the operation $\overline{t}^{\alpha}$ in $\rightsquigarrow_{\mathtt{e}_{red}}$ and $\rightsquigarrow_{\mathtt{e}_{abs}}$ in Fig. 4), but this does not affect the overall complexity, as it speeds up other operations, see [1]. The entries of E are of the form $[x{\leftarrow}\overline{t}]^l$, *i.e.* they carry a label l used to implement usefulness, to be explained later on in this section. We write $E(x) = [x{\leftarrow}\overline{t}]^l$ when E contains $[x{\leftarrow}\overline{t}]^l$ and $E(x) = \bot$ when in E there are no entries of the form $[x{\leftarrow}\overline{t}]^l$.

The Decoding Every state s decodes to a term $\underline{s}$ (see Fig. 3), having the shape $C_s\langle\overline{t}{\downarrow_E}\rangle$, where

- $\overline{t}{\downarrow_E}$ is a λ-term, roughly obtained by applying to the code the substitution induced by the global environment E. More precisely, the operation $\overline{t}{\downarrow_E}$ is called *unfolding* and it is properly defined at the end of this section.
- C_s is a context, that will be shown to be a LO context, obtained by decoding the stack π and the dump F and applying the unfolding. Note that, to improve readability, π is decoded in postfix notation for plugging.

The Transitions According to the distillation approach of [1] we distinguish different kinds of transitions, whose names reflect a proof-theoretical view, as machine transitions can be seen as cut-elimination steps [7,1]:

- *Multiplicatives* $\leadsto_{\mathtt{m}}$: they fire a β-redex, except that if the argument is not a variable then it is not substituted but added to the environment;
- *Exponentials* $\leadsto_{\mathtt{e}}$: they perform a clashing-avoiding substitution from the environment on the single variable occurrence represented by the current code. They implement micro-step substitution.
- *Commutatives* $\leadsto_{\mathtt{c}}$: they locate and expose the next redex according to the LO evaluation strategy, by rearranging the data-structures.

Both exponential and commutative transitions are invisible on the λ-calculus. Garbage collection is here simply ignored, or, more precisely, it is encapsulated at the meta-level, in the decoding function.

Labels for Useful Sharing A label l for a code in the environment can be of three kinds. Roughly, they are:

- *Neutral*, or $l = neu$: it marks a neutral term, that is always useless as it is β-normal and its substitution cannot create a redex, because it is not an abstraction;
- *Abstraction*, or $l = abs$: it marks an abstraction, that is a term that is at times useful to substitute. If the variable that it is meant to replace is applied, indeed, the substitution of the abstraction creates a β-redex. But if it is not applied, it is useless.
- *Redex*, or $l = red$: it marks a term that contains a β-redex. It is always useful to substitute these terms.

Actually, the explanation we just gave is oversimplified, but it provides a first intuition about labels. In fact in an environment $[x{\leftarrow}\overline{t}]^l : E$ it is not really $\overline{t}$ that has the property mentioned by its label, rather the term $\overline{t}{\downarrow}_E$ obtained by unfolding the rest of the environment on $\overline{t}$. The idea is that $[x{\leftarrow}\overline{t}]^{red}$ states that it is useful to substitute $\overline{t}$ to *later on* obtain a redex inside it (by potential further substitutions on its variables coming from E). The precise meaning of the labels will be given by Definition 6, and the properties they encode will be made explicit by Lemma 11.

A further subtlety is that the label *red* for redexes is refined as a pair (red, n), where n is the number of substitutions in E that are needed to obtain the LO redex in $\overline{t}{\downarrow}_E$. Our machines never inspect these numbers, they are only used for the complexity analysis of Sect. 5.2.

Grafting and Unfoldings The unfolding of the environment E on a code $\overline{t}$ is defined as the recursive *capture-allowing* substitution (called *grafting*) of the entries of E on $\overline{t}$.

Definition 5 (Grafting and Environment Unfolding). *The operation of grafting $\overline{t}\{\{x{\leftarrow}\overline{u}\}\}$ is defined by*

$$(\overline{wr})\{\{x{\leftarrow}\overline{u}\}\} := \overline{w}\{\{x{\leftarrow}\overline{u}\}\}\overline{r}\{\{x{\leftarrow}\overline{u}\}\} \qquad (\lambda y.\overline{w})\{\{x{\leftarrow}\overline{u}\}\} := \lambda y.\overline{w}\{\{x{\leftarrow}\overline{u}\}\}$$
$$x\{\{x{\leftarrow}\overline{u}\}\} := \overline{u} \qquad y\{\{x{\leftarrow}\overline{u}\}\} := y$$

Frames	$F ::= \epsilon \mid F : \phi$	Stacks	$\pi ::= \epsilon \mid \bar{t} : \pi$
Frame Items	$\phi ::= \bar{t}\Diamond\pi \mid x$	Phases	$\varphi ::= \blacktriangledown \mid \blacktriangle$
Labels	$l ::= abs \mid (red, n \in \mathbb{N}) \mid neu$	Environments	$E ::= \epsilon \mid [x{\leftarrow}\bar{t}]^l : E$

Fig. 1: Grammars.

Given an environment E we define the unfolding of E on a code $\bar{t}$ as follows:

$$\bar{t}{\downarrow}_{\epsilon} := \bar{t} \qquad \bar{t}{\downarrow}_{[x\leftarrow\bar{u}]^l:E} := \bar{t}\{\{x{\leftarrow}\bar{u}\}\}{\downarrow}_E$$

or equivalently as:

$$\begin{array}{rlll} (\overline{uw}){\downarrow}_E := \bar{u}{\downarrow}_E \bar{w}{\downarrow}_E & x{\downarrow}_{[x\leftarrow\bar{u}]^l:E'} := \bar{u}{\downarrow}_{E'} & \\ (\lambda x.\bar{u}){\downarrow}_E := \lambda x.\bar{u}{\downarrow}_E & x{\downarrow}_{[y\leftarrow\bar{u}]^l:E'} := x{\downarrow}_{E'} & x{\downarrow}_{\epsilon} := x \end{array}$$

For instance, $(\lambda x.y){\downarrow}_{[y\leftarrow xx]^{neu}} = \lambda x.(xx)$. The unfolding is extended to contexts as expected (*i.e.* recursively propagating the unfolding and setting $\langle\cdot\rangle{\downarrow}_E = E$).

Let us explain the need for grafting. In [2], the Strong MAM is decoded to the LSC, that is a calculus with explicit substitutions, *i.e.* a calculus able to represent the environment of the Strong MAM. Matching the representation of the environment on the Strong MAM and on the LSC does not need grafting but it is, however, a quite technical affair. Useful sharing adds many further complications in establishing such a matching, because useful evaluation computes a shared representation of the normal form and forces some of the explicit substitutions to stay under abstractions. The difficulty is such, in fact, that we found much easier to decode directly to the λ-calculus rather than to the LSC. Such an alternative solution, however, has to push the substitution induced by the environment through abstractions, which is why we use grafting.

Lemma 3 (Properties of Grafting and Unfolding).

1. *If the bound names of t do not appear free in u then $t\{x{\leftarrow}u\} = t\{\{x{\leftarrow}u\}\}$.*
2. *If moreover they do not appear free in E then $t{\downarrow}_E\{x{\leftarrow}u{\downarrow}_E\} = t\{x{\leftarrow}u\}{\downarrow}_E$.*

4 The Checking Abstract Machine

The Checking Abstract Machine (Checking AM) is defined in Fig. 2. It starts executions on states of the form $(\epsilon, \bar{t}, \epsilon, E, \blacktriangledown)$, with the aim of checking the usefulness of $\bar{t}$ with respect to the environment E, *i.e.* it walks through $\bar{t}$ and whenever it encounters a variable x it looks up its usefulness in E.

The Checking AM has six commutative transitions, noted $\rightharpoonup_{\mathtt{c}_i}$ with $i = 1, .., 6$, used to walk through the term, and five output transitions, noted $\rightharpoonup_{\mathtt{o}_j}$ with $j = 1, .., 5$, that produce the value of the test for usefulness, to be later used by the Useful MAM. The exploration is done in two alternating phases, evaluation $\blacktriangledown$ and backtracking $\blacktriangle$. Evaluation explores the current code towards

Frame	Code	Stack	Env	Ph		Frame	Code	Stack	Env	Ph
F	$\overline{t}\overline{u}$	π	E	▼	$\rightharpoonup_{\blacktriangledown\mathtt{c}_1}$	F	$\overline{t}$	$\overline{u} : \pi$	E	▼
F	$\lambda x.\overline{t}$	$\overline{u} : \pi$	E	▼	$\rightharpoonup_{\mathtt{o}_1}$	output $(red, 1)$				
F	$\lambda x.\overline{t}$	ϵ	E	▼	$\rightharpoonup_{\blacktriangledown\mathtt{c}_2}$	$F : x$	$\overline{t}$	ϵ	E	▼
F	x	π	E	▼	$\rightharpoonup_{\mathtt{o}_2}$	output $(red, n+1)$				
						if $E(x) = [x{\leftarrow}\overline{t}]^{(red,n)}$				
F	x	$\overline{u} : \pi$	E	▼	$\rightharpoonup_{\mathtt{o}_3}$	output $(red, 2)$				
						if $E(x) = [x{\leftarrow}\overline{t}]^{abs}$				
F	x	π	E	▼	$\rightharpoonup_{\blacktriangledown\mathtt{c}_3}$	F	x	π	E	▲
		if $E(x) = \bot$ or $E(x) = [x{\leftarrow}\overline{t}]^{neu}$ or $(E(x) = [x{\leftarrow}\overline{t}]^{abs}$ and $\pi = \epsilon)$								
$F : x$	$\overline{t}$	ϵ	E	▲	$\rightharpoonup_{\blacktriangle\mathtt{c}_4}$	F	$\lambda x.\overline{t}$	ϵ	E	▲
$F : \overline{t}\Diamond\pi$	$\overline{u}$	ϵ	E	▲	$\rightharpoonup_{\blacktriangle\mathtt{c}_5}$	F	$\overline{t}\overline{u}$	π	E	▲
F	$\overline{t}$	$\overline{u} : \pi$	E	▲	$\rightharpoonup_{\blacktriangle\mathtt{c}_6}$	$F : \overline{t}\Diamond\pi$	$\overline{u}$	ϵ	E	▼
ϵ	$\overline{t}\overline{u}$	ϵ	E	▲	$\rightharpoonup_{\mathtt{o}_4}$	output neu				
ϵ	$\lambda x.\overline{t}$	ϵ	E	▲	$\rightharpoonup_{\mathtt{o}_5}$	output abs				

Fig. 2: The Checking Abstract Machine (Checking AM).

$$\begin{aligned}
\underline{\epsilon} &:= \langle\cdot\rangle & \underline{C_s} &:= \underline{F}\langle\underline{\pi}\rangle{\downarrow_E}\\
\underline{\overline{u} : \pi} &:= \langle\langle\cdot\rangle\overline{u}\rangle\underline{\pi} & \underline{s} &:= \underline{F}\langle\langle\overline{t}\rangle\underline{\pi}\rangle{\downarrow_E} = \underline{C_s}\langle\overline{t}{\downarrow_E}\rangle\\
\underline{F : \overline{t}\Diamond\pi} &:= \underline{F}\langle\langle\overline{t}\langle\cdot\rangle\rangle\underline{\pi}\rangle & &\\
\underline{F : x} &:= \underline{F}\langle\lambda x.\langle\cdot\rangle\rangle & &\text{where } s = (F, \overline{t}, \pi, E)
\end{aligned}$$

Fig. 3: Decoding.

the head, storing in the stack and in the frame the parts of the code that it leaves behind. Backtracking comes back to an argument that was stored in the frame, when the current head has already been checked. Note that the Checking AM never modifies the environment, it only looks it up.

Let us explain the transitions. First the commutative ones:

- $\rightharpoonup_{\blacktriangledown\mathtt{c}_1}$: the code is an application $\overline{t}\overline{u}$ and the machine starts exploring the left subterm $\overline{t}$, storing $\overline{u}$ on top of the stack π.
- $\rightharpoonup_{\blacktriangledown\mathtt{c}_2}$: the code is an abstraction $\lambda x.\overline{t}$ and the machine goes under the abstraction, storing x on top of the frame F.
- $\rightharpoonup_{\blacktriangledown\mathtt{c}_3}$: the machine finds a variable x that either has no associated entry in the environment (if $E(x) = \bot$) or its associated entry $[x{\leftarrow}\overline{t}]^l$ in the environment is useless. This can happen if either $l = neu$, *i.e.* substituting $\overline{t}$ would only lead to a neutral term, or $l = abs$, *i.e.* substituting $\overline{t}$ would provide an abstraction, but the stack is empty, and so it is useless to substitute the abstraction because no β-redexes will be obtained. Thus the machine switches to the backtracking phase (▲), whose aim is to undo the frame to obtain a new subterm to explore.
- $\rightharpoonup_{\blacktriangle\mathtt{c}_4}$: it is the inverse of $\rightharpoonup_{\blacktriangledown\mathtt{c}_2}$, it puts back on the code an abstraction that was previously stored in the frame.
- $\rightharpoonup_{\blacktriangle\mathtt{c}_5}$: backtracking from the evaluation of an argument $\overline{u}$, it restores the application $\overline{t}\overline{u}$ and the stack π that were previously stored in the frame.

– $\rightharpoonup_{\blacktriangle \mathtt{c}_6}$: backtracking from the evaluation of the left subterm $\overline{t}$ of an application $\overline{t}\overline{u}$, the machine starts evaluating the right subterm (by switching to the evaluation phase $\blacktriangledown$) with an empty stack ϵ, storing on the frame the pair $\overline{t}\lozenge\pi$ of the left subterm and the previous stack π.

Then the output transitions:

– $\rightharpoonup_{\mathtt{o}_1}$: the machine finds a β-redex, namely $(\lambda x.\overline{t})\overline{u}$ and thus outputs a label saying that it requires only one substitution step (namely substituting the term the machine was executed on) to eventually find a β-redex.
– $\rightharpoonup_{\mathtt{o}_2}$: the machine finds a variable x whose associated entry $[x{\leftarrow}\overline{t}]^{(red,n)}$ in the environment is labeled with (red, n), and so outputs a label saying that it takes $n + 1$ substitution steps to eventually find a β-redex (n plus 1 for the term the machine was executed on).
– $\rightharpoonup_{\mathtt{o}_3}$: the machine finds a variable x whose associated entry $[x{\leftarrow}\overline{t}]^{abs}$ in the environment is labeled with abs, so $\overline{t}$ is an abstraction, and the stack is non-empty. Since substituting the abstraction will create a β-redex, the machine outputs a label saying that it takes two substitution steps to obtain a β-redex, one for the term the machine was executed on and one for the abstraction $\overline{t}$.
– $\rightharpoonup_{\mathtt{o}_4}$: the machine went through the whole term, that is an application, and found no redex, nor any redex that can be obtained by substituting from the environment. Thus that term is neutral and so the machine outputs the corresponding label.
– $\rightharpoonup_{\mathtt{o}_5}$: as for the previous transition, except that the term is an abstraction, and so the output is the abs label.

The fact that commutative transitions only walk through the code, without changing anything, is formalised by the following lemma, that is crucial for the proof of correctness of the Checking AM (forthcoming Theorem 1).

Lemma 4 (Commutative Transparency).
Let $s = (F, \overline{u}, \pi, E, \varphi) \rightsquigarrow_{\mathtt{c}_{1,2,3,4,5,6}} (F', \overline{u}', \pi', E, \varphi') = s'$. *Then*

1. Decoding Without Unfolding*:* $\underline{F}\langle\langle\overline{u}\rangle\underline{\pi}\rangle = \underline{F'}\langle\langle\overline{u}'\rangle\underline{\pi'}\rangle$*, and*
2. Decoding With Unfolding*:* $\underline{s} = \underline{s'}$*.*

For the analysis of the properties of the Checking AM we need a notion of well-labeled environment, *i.e.* of environment where the labels are consistent with their intended meaning. It is a technical notion also providing enough information to perform the complexity analysis, later on. Moreover, it includes two structural properties of environments: 1) in $[x{\leftarrow}\overline{t}]^l$ the code $\overline{t}$ cannot be a variable, and 2) there cannot be two entries associated to the same variables.

Definition 6 (Well-Labeled Environments). Well-labeled global environments E *are defined by*

1. Empty*:* ϵ *is well-labeled;*
2. Inductive*:* $[x{\leftarrow}\overline{t}]^l : E'$ *is well-labeled if* E' *is well-labeled,* x *is fresh with respect to* $\overline{t}$ *and* E'*, and*

(a) Abstractions*: if $l = abs$ then $\overline{t}$ and $\overline{t}\downarrow_{E'}$ are normal abstractions;*

(b) Neutral Terms*: if $l = neu$ then $\overline{t}$ is an application and $\overline{t}\downarrow_{E'}$ is neutral.*

(c) Redexes*: if $l = (red, n)$ then $\overline{t}$ is not a variable, $\overline{t}\downarrow_{E'}$ contains a β-redex. Moreover, $\overline{t} = C\langle\overline{u}\rangle$ with C a LO context and*
   - *if $n = 1$ then $\overline{u}$ is a β-redex,*
   - *if $n > 1$ then $\overline{u} = x$ and $E' = E'' : [y{\leftarrow}\overline{u}]^l : E'''$ with*
     - *if $n > 2$ then $l = (red, n - 1)$*
     - *if $n = 2$ then $l = (red, 1)$ or ($l = abs$ and C is applicative).*

Remark 1. Note that by the definition it immediately follows that if $E = E' : [x{\leftarrow}\overline{t}]^{(red,n)} : E''$ is well-labeled then the length of E'', and thus of E, is at least n. This fact is used in the proof of Theorem 3.1.

The study of the Checking AM requires some terminology and two invariants. A state s is *initial* if it is of the form $(\epsilon, \overline{t}, \epsilon, E, \varphi)$ with E well-labeled and it is *reachable* if there are an initial state s' and a Checking AM execution $\rho : s' \rightharpoonup^* s$. Both invariants are used to prove the correctness of the Checking AM: the *normal form invariant* to guarantee that codes labeled with *neu* and *abs* are indeed normal or neutral, while the *decoding invariant* is used for the redex labels.

Lemma 5 (Checking AM Invariants). *Let $s = F \mid \overline{u} \mid \pi \mid E \mid \varphi$ be a Checking AM reachable state and E be a well-labeled environment.*

1. Normal Form*:*
   (a) Backtracking Code*: if $\varphi = \blacktriangle$, then $\overline{u}\downarrow_E$ is normal, and if π is non-empty, then $\overline{u}\downarrow_E$ is neutral;*
   (b) Frame*: if $F = F' : \overline{w}\Diamond\pi' : F''$, then $\overline{w}\downarrow_E$ is neutral.*
2. Decoding*: C_s is a LO context.*

Finally, we can prove the main properties of the Checking AM, *i.e.* that when executed on $\overline{t}$ and E it provides a label l to extend E with a consistent entry for $\overline{t}$ (*i.e.* such that $[x{\leftarrow}\overline{t}]^l : E$ is well-labeled), and that such an execution takes time linear in the size of $\overline{t}$.

Theorem 1 (Checking AM Properties). *Let $\overline{t}$ be a code and E a global environment.*

1. Determinism and Progress*: the Checking AM is deterministic and there always is a transition that applies;*
2. Termination and Complexity*: the execution of the Checking AM on $\overline{t}$ and E always terminates, taking $O(|\overline{t}|)$ steps, moreover*
3. Correctness*: if E is well-labeled, x is fresh with respect to E and $\overline{t}$, and l is the output then $[x{\leftarrow}\overline{t}]^l : E$ is well-labeled.*

Frame	Code	Stack	Env	Ph		Frame	Code	Stack	Env	Ph
F	$\overline{t}\overline{u}$	π	E	▼	$\rightsquigarrow_{▼\mathtt{c}_1}$	F	$\overline{t}$	$\overline{u}:\pi$	E	▼
F	$\lambda x.\overline{t}$	$y:\pi$	E	▼	$\rightsquigarrow_{\mathtt{m}_1}$	F	$\overline{t}\{x{\leftarrow}y\}$	π	E	▼
F	$\lambda x.\overline{t}$	$\overline{u}:\pi$	E	▼	$\rightsquigarrow_{\mathtt{m}_2}$	F	$\overline{t}$	π	$[x{\leftarrow}\overline{u}]^l : E$	▼
									if $\overline{u}$ is not a variable and l is the output of the Checking AM on $\overline{u}$ and E	
F	$\lambda x.\overline{t}$	ϵ	E	▼	$\rightsquigarrow_{▼\mathtt{c}_2}$	$F:x$	$\overline{t}$	ϵ	E	▼
F	x	π	E	▼	$\rightsquigarrow_{\mathtt{e}_{red}}$	F	$\overline{t}^\alpha$	π	E	▼
									if $E(x) = [x{\leftarrow}\overline{t}]^{(red,n)}$	
F	x	$\overline{u}:\pi$	E	▼	$\rightsquigarrow_{\mathtt{e}_{abs}}$	F	$\overline{t}^\alpha$	$\overline{u}:\pi$	E	▼
									if $E(x) = [x{\leftarrow}\overline{t}]^{abs}$	
F	x	π	E	▼	$\rightsquigarrow_{▼\mathtt{c}_3}$	F	x	π	E	▲
									if $E(x) = \bot$ or $E(x) = [x{\leftarrow}\overline{t}]^{neu}$ or ($E(x) = [x{\leftarrow}\overline{t}]^{abs}$ and $\pi = \epsilon$)	
$F:x$	$\overline{t}$	ϵ	E	▲	$\rightsquigarrow_{▲\mathtt{c}_4}$	F	$\lambda x.\overline{t}$	ϵ	E	▲
$F:\overline{t}\Diamond\pi$	$\overline{u}$	ϵ	E	▲	$\rightsquigarrow_{▲\mathtt{c}_5}$	F	$\overline{t}\overline{u}$	π	E	▲
F	$\overline{t}$	$\overline{u}:\pi$	E	▲	$\rightsquigarrow_{▲\mathtt{c}_6}$	$F:\overline{t}\Diamond\pi$	$\overline{u}$	ϵ	E	▼

$\overline{t}^\alpha$ is any code α-equivalent to $\overline{t}$ such that it is well-named and its bound names are fresh with respect to those in the other machine components.

Fig. 4: The Useful Milner Abstract Machine (Useful MAM).

5 The Useful Milner Abstract Machine

The Useful MAM is defined in Fig. 4. It is very similar to the Checking AM, in particular it has exactly the same commutative transitions, and the same organisation in evaluating and backtracking phases. The difference with respect to the Useful MAM is that the output transitions are replaced by micro-step computational rules that reduce β-redexes and implement useful substitutions. Let us explain them:

- *Multiplicative Transition* $\rightsquigarrow_{\mathtt{m}_1}$: when the argument of the β-redex $(\lambda x.\overline{t})y$ is a variable y then it is immediately substituted in $\overline{t}$. This happens because 1) such substitution are not costly and 2) because in this way the environment stays compact, see also Remark 2 at the end of the paper.
- *Multiplicative Transition* $\rightsquigarrow_{\mathtt{m}_2}$: if the argument $\overline{u}$ is not a variable then the entry $[x{\leftarrow}\overline{u}]^l$ is added to the environment. The label l is obtained by running the Checking AM on $\overline{u}$ and E.
- *Exponential Transition* $\rightsquigarrow_{\mathtt{e}_{red}}$: the environment entry associated to x is labeled with (red, n) thus it is useful to substitute $\overline{t}$. The idea is that in at most n additional substitution steps (shuffled with commutative steps) a β-redex will be obtained. To avoid variable clashes the substitution α-renames $\overline{t}$.
- *Exponential Transition* $\rightsquigarrow_{\mathtt{e}_{abs}}$: the environment associates an abstraction to x and the stack is non empty, so it is useful to substitute the abstraction (again, α-renaming to avoid variable clashes). Note that if the stack is empty the machine rather backtracks using $\rightsquigarrow_{▼\mathtt{c}_3}$.

The Useful MAM starts executions on *initial states* of the form $(\epsilon, \overline{t}, \epsilon, \epsilon)$, where $\overline{t}$ is such that any two variables (bound or free) have distinct names, and any other component is empty. A state s is *reachable* if there are an initial state s' and a Useful MAM execution $\rho : s' \rightsquigarrow^* s$, and it is *final* if no transitions apply.

5.1 Qualitative Analysis

The results of this subsection are the correctness and completeness of the Useful MAM. Four invariants are required. The *normal form* and *decoding invariants* are exactly those of the Checking AM (and the proof for the commutative transitions is the same). The *environment labels invariant* follows from the correctness of the Checking AM (Theorem 1.2). The *name invariant* is used in the proof of Lemma 7.

Lemma 6 (Useful MAM Qualitative Invariants). *Let $s = F \mid \overline{u} \mid \pi \mid E \mid \varphi$ be a state reachable from an initial term $\overline{t}_0$. Then:*

1. Environment Labels*: E is well-labeled.*
2. Normal Form*:*
   (a) Backtracking Code*: if $\varphi = \blacktriangle$, then $\overline{u}{\downarrow_E}$ is normal, and if π is non-empty, then $\overline{u}{\downarrow_E}$ is neutral;*
   (b) Frame*: if $F = F' : \overline{w}\lozenge\pi' : F''$, then $\overline{w}{\downarrow_E}$ is neutral.*
3. Name:
   (a) Substitutions*: if $E = E' : [x{\leftarrow}\overline{t}] : E''$ then x is fresh wrt $\overline{t}$ and E'';*
   (b) Abstractions and Evaluation*: if $\varphi = \blacktriangledown$ and $\lambda x.\overline{t}$ is a subterm of $\overline{u}$, π, or π' (if $F = F' : \overline{w}\lozenge\pi' : F''$) then x may occur only in $\overline{t}$;*
   (c) Abstractions and Backtracking*: if $\varphi = \blacktriangle$ and $\lambda x.\overline{t}$ is a subterm of π or π' (if $F = F' : \overline{w}\lozenge\pi' : F''$) then x may occur only in $\overline{t}$.*
4. Decoding*: C_s is a LO context.*

We can now show how every single transition projects on the λ-calculus, and in particular that multiplicative transitions project to LO β-steps.

Lemma 7 (One-Step Weak Simulation, Proof at Page 17). *Let s be a reachable state.*

1. Commutative*: if $s \leadsto_{\mathtt{c}_{1,2,3,4,5,6}} s'$ then $\underline{s} = \underline{s'}$;*
2. Exponential*: if $s \leadsto_{\mathtt{e}_{red},\mathtt{e}_{abs}} s'$ then $\underline{s} = \underline{s'}$;*
3. Multiplicative*: if $s \leadsto_{\mathtt{m}_1,\mathtt{m}_2} s'$ then $\underline{s} \to_{\mathtt{LO}\beta} \underline{s'}$.*

We also need to show that the Useful MAM computes β-normal forms.

Lemma 8 (Progress, Proof at Page 18). *Let s be a reachable final state. Then $\underline{s}$ is β-normal.*

The theorem of correctness and completeness of the machine with respect to $\to_{\mathtt{LO}\beta}$ follows. The bisimulation is *weak* because transitions other than $\leadsto_{\mathtt{m}}$ are invisible on the λ-calculus. For a machine execution ρ we denote with $|\rho|$ (resp. $|\rho|_{\mathtt{x}}$) the number of transitions (resp. $\mathtt{x}$-transitions for $\mathtt{x} \in \{\mathtt{m}, \mathtt{e}, \mathtt{c}, \ldots\}$) in ρ.

Theorem 2 (Weak Bisimulation, Proof at Page 18). *Let s be an initial Useful MAM state of code $\overline{t}$.*

1. Simulation: *for every execution $\rho : s \leadsto^* s'$ there exists a derivation $d\colon \underline{s} \to^*_{\mathtt{LO}\beta} \underline{s'}$ such that $|d| = |\rho|_{\mathtt{m}}$;*
2. Reverse Simulation: *for every derivation $d\colon \overline{t} \to^*_{\mathtt{LO}\beta} u$ there is an execution $\rho : s \leadsto^* s'$ such that $\underline{s'} = u$ and $|d| = |\rho|_{\mathtt{m}}$.*

5.2 Quantitative Analysis

The complexity analyses of this section rely on two additional invariants of the Useful MAM, the subterm and the environment size invariants.

The subterm invariant bounds the size of the duplicated subterms and it is crucial. For us, $\overline{u}$ is a subterm of $\overline{t}$ if it does so up to variable names, both free and bound. More precisely: define t^- as t in which all variables (including those appearing in binders) are replaced by a fixed symbol $*$. Then, we will consider u to be a subterm of t whenever u^- is a subterm of t^- in the usual sense. The key property ensured by this definition is that the size $|\overline{u}|$ of $\overline{u}$ is bounded by $|\overline{t}|$.

Lemma 9 (Useful MAM Quantitative Invariants). *Let $s = F \mid \overline{u} \mid \pi \mid E \mid \varphi$ be a state reachable by the execution ρ from the initial code $\overline{t}_0$.*

1. Subterm*:*
   - *(a)* Evaluating Code*: if $\varphi = ▼$, then $\overline{u}$ is a subterm of $\overline{t}_0$;*
   - *(b)* Stack*: any code in the stack π is a subterm of $\overline{t}_0$;*
   - *(c)* Frame*: if $F = F' : \overline{w}\Diamond\pi' : F''$, then any code in π' is a subterm of $\overline{t}_0$;*
   - *(d)* Global Environment*: if $E = E' : [x{\leftarrow}\overline{w}]^l : E''$, then $\overline{w}$ is a subterm of $\overline{t}_0$;*
2. Environment Size*: the length of the global environment E is bound by $|\rho|_{\mathtt{m}}$.*

The proof of the polynomial bound of the overhead is in three steps. First, we bound the number $|\rho|_{\mathtt{e}}$ of exponential transitions of an execution ρ using the number $|\rho|_{\mathtt{m}}$ of multiplicative transitions of ρ, that by Theorem 2 corresponds to the number of LO β-steps on the λ-calculus. Second, we bound the number $|\rho|_c$ of commutative transitions of ρ by using the number of exponential transitions and the size of the initial term. Third, we put everything together.

Multiplicative vs Exponential Analysis This step requires two auxiliary lemmas. The first one essentially states that commutative transitions *eat* normal and neutral terms, as well as LO contexts.

Lemma 10. *Let $s = F \mid \overline{t} \mid \pi \mid E \mid ▼$ be a state and E be well-labeled. Then*

1. *If $\overline{t}{\downarrow_E}$ is a normal term and $\pi = \epsilon$ then $s \leadsto_{\mathtt{c}}^* F \mid \overline{t} \mid \pi \mid E \mid ▲$.*
2. *If $\overline{t}{\downarrow_E}$ is a neutral term then $s \leadsto_{\mathtt{c}}^* F \mid \overline{t} \mid \pi \mid E \mid ▲$.*
3. *If $\overline{t} = C\langle\overline{u}\rangle$ with $C{\downarrow_E}$ a LO context then there exist F' and π' such that $s \leadsto_{\mathtt{c}}^* F' \mid \overline{u} \mid \pi' \mid E \mid ▼$;*

The second lemma uses Lemma 10 and the environment labels invariant (Lemma 6.1) to show that the exponential transitions of the Useful MAM are indeed useful, as they head towards a multiplicative transition, that is towards β-redexes.

Lemma 11 (Useful Exponentials Lead to Multiplicatives). *Let s be a reachable state such that $s \leadsto_{\mathtt{e}_{(red,n)}} s'$.*

1. *If $n = 1$ then $s' \leadsto_{\mathtt{c}}^* \leadsto_{\mathtt{m}} s''$;*

2. *If $n = 2$ then $s' \rightsquigarrow_{\mathtt{c}}^{*} \rightsquigarrow_{\mathtt{e}_{abs}} \rightsquigarrow_{\mathtt{m}} s''$ or $s' \rightsquigarrow_{\mathtt{c}}^{*} \rightsquigarrow_{\mathtt{e}_{(red,1)}} s''$;*
3. *If $n > 1$ then $s' \rightsquigarrow_{\mathtt{c}}^{*} \rightsquigarrow_{\mathtt{e}_{(red,n-1)}} s''$.*

Finally, using the environment size invariant (Lemma 9.2) we obtain the local boundedness property, that is used to infer a quadratic bound via a standard reasoning (already employed in [6]).

Theorem 3 (Exponentials vs Multiplicatives, Proof at Page 19). *Let s be an initial Useful MAM state and $\rho : s \rightsquigarrow^{*} s'$.*

1. Local Boundedness*: if $\sigma : s' \rightsquigarrow^{*} s''$ and $|\sigma|_{\mathtt{m}} = 0$ then $|\sigma|_{\mathtt{e}} \leq |\rho|_{\mathtt{m}}$;*
2. Exponentials are Quadratic in the Multiplicatives*: $|\rho|_{\mathtt{e}} \in O(|\rho|_{\mathtt{m}}^{2})$.*

Commutative vs Exponential Analysis The second step is to bound the number of commutative transitions. Since the commutative part of the Useful MAM is essentially the same as the commutative part of the Strong MAM of [2], the proof of such bound is essentially the same as in [2]. It relies on the subterm invariant (Lemma 9.1).

Theorem 4 (Commutatives vs Exponentials, Proof at Page 20). *Let $\rho : s \rightsquigarrow^{*} s'$ be a Useful MAM execution from an initial state of code t. Then:*

1. Commutative Evaluation Steps are Bilinear*: $|\rho|_{\blacktriangledown \mathtt{c}} \leq (1 + |\rho|_{\mathtt{e}}) \cdot |t|$.*
2. Commutative Evaluation Bounds Backtracking*: $|\rho|_{\blacktriangle \mathtt{c}} \leq 2 \cdot |\rho|_{\blacktriangledown \mathtt{c}}$.*
3. Commutative Transitions are Bilinear*: $|\rho|_{c} \leq 3 \cdot (1 + |\rho|_{\mathtt{e}}) \cdot |t|$.*

The Main Theorem Putting together the matching between LO β-steps and multiplicative transitions (Theorem 2), the quadratic bound on the exponentials via the multiplicatives (Theorem 3.2) and the bilinear bound on the commutatives (Theorem 4.3) we obtain that the number of the Useful MAM transitions to implement a LO β-derivation d is at most quadratic in the length of d and linear in the size of t. Moreover, the subterm invariant (Lemma 9.1) and the analysis of the Checking AM (Theorem 1.2) allow to bound the cost of implementing the execution on RAM.

Theorem 5 (Useful MAM Overhead Bound, Proof at Page 20). *Let $d : t \rightarrow_{\mathtt{LO}\beta}^{*} u$ be a leftmost-outermost derivation and ρ be the Useful MAM execution simulating d given by Theorem 2.2. Then:*

1. Length*: $|\rho| = O((1 + |d|^{2}) \cdot |t|)$.*
2. Cost*: ρ is implementable on RAM in $O((1 + |d|^{2}) \cdot |t|)$ steps.*

Remark 2. Our bound is quadratic in the number of the LO β-steps but we believe that it is not tight. In fact, our transition $\rightsquigarrow_{\mathtt{m}_1}$ is a standard optimisation, used for instance in Wand's [28] (section 2), Friedman et al.'s [21] (section 4), and Sestoft's [26] (section 4), and motivated as an optimization about *space*. In Sands, Gustavsson, and Moran's [25], however, it is shown that it lowers the overhead for *time* from quadratic to linear (with respect to the number of β-steps) for call-by-name evaluation in a weak setting. Unfortunately, the simple proof used in [25] does not scale up to our setting, nor we have an alternative proof that the overhead is linear. We conjecture, however, that it does.

References

1. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling Abstract Machines. In: ICFP 2014. pp. 363–376 (2014)
2. Accattoli, B., Barenbaum, P., Mazza, D.: A Strong Distillery. In: APLAS 2015. pp. 231–250 (2015)
3. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A Nonstandard Standardization Theorem. In: POPL. pp. 659–670 (2014)
4. Accattoli, B., Coen, C.S.: On the Relative Usefulness of Fireballs. In: LICS 2015. pp. 141–155 (2015)
5. Accattoli, B., Dal Lago, U.: On the invariance of the unitary cost model for head reduction. In: RTA. pp. 22–37 (2012)
6. Accattoli, B., Lago, U.D.: (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. Logical Methods in Computer Science 12(1) (2016)
7. Ariola, Z.M., Bohannon, A., Sabry, A.: Sequent calculi and abstract machines. ACM Trans. Program. Lang. Syst. 31(4) (2009)
8. Asperti, A., Mairson, H.G.: Parallel beta reduction is not elementary recursive. In: POPL. pp. 303–315 (1998)
9. Blelloch, G.E., Greiner, J.: Parallelism in sequential functional languages. In: FPCA. pp. 226–237 (1995)
10. Boutiller, P.: De nouveaus outils pour manipuler les inductif en Coq. Ph.D. thesis, Université Paris Diderot - Paris 7 (2014)
11. de Carvalho, D.: Execution time of lambda-terms via denotational semantics and intersection types. CoRR abs/0905.4251 (2009)
12. Crégut, P.: An abstract machine for lambda-terms normalization. In: LISP and Functional Programming. pp. 333–340 (1990)
13. Crégut, P.: Strongly reducing variants of the Krivine abstract machine. Higher-Order and Symbolic Computation 20(3), 209–230 (2007)
14. Dal Lago, U., Martini, S.: The weak lambda calculus as a reasonable machine. Theor. Comput. Sci. 398(1-3), 32–50 (2008)
15. Danos, V., Regnier, L.: Head linear reduction. Tech. rep. (2004)
16. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Tech. Rep. RS-04-26, BRICS (2004)
17. Danvy, O., Zerny, I.: A synthetic operational account of call-by-need evaluation. In: PPDP. pp. 97–108 (2013)
18. Dénès, M.: Étude formelle d'algorithmes efficaces en algèbre linéaire. Ph.D. thesis, Université de Nice - Sophia Antipolis (2013)
19. Ehrhard, T., Regnier, L.: Böhm trees, Krivine's machine and the Taylor expansion of lambda-terms. In: CiE. pp. 186–197 (2006)
20. Fernández, M., Siafakas, N.: New developments in environment machines. Electr. Notes Theor. Comput. Sci. 237, 57–73 (2009)
21. Friedman, D.P., Ghuloum, A., Siek, J.G., Winebarger, O.L.: Improving the lazy krivine machine. Higher-Order and Symbolic Computation 20(3), 271–293 (2007)
22. García-Pérez, Á., Nogueira, P., Moreno-Navarro, J.J.: Deriving the full-reducing krivine machine from the small-step operational semantics of normal order. In: PPDP. pp. 85–96 (2013)
23. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: (ICFP '02). pp. 235–246 (2002)
24. Milner, R.: Local bigraphs and confluence: Two conjectures. Electr. Notes Theor. Comput. Sci. 175(3), 65–73 (2007)

25. Sands, D., Gustavsson, J., Moran, A.: Lambda calculi and linear speedups. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. pp. 60–84 (2002)
26. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. 7(3), 231–264 (1997)
27. Smith, C.: Abstract machines for higher-order term sharing, Presented at IFL 2014
28. Wand, M.: On the correctness of the krivine machine. Higher-Order and Symbolic Computation 20(3), 231–235 (2007)

Proofs of the Main Lemmas and Theorems

Proof of One-Step Weak Bisimulation Lemma (Lemma 7, p. 13)

1. *Commutative*: the proof is exactly as the one for the Checking AM (Lemma 4.2), that can be found in the longer version of this paper on the author's webpage.
2. *Exponential*:
   - **Case** $s = (F, x, \pi, E, \blacktriangledown) \rightsquigarrow_{\mathtt{e}_{red}} (F, \overline{t}^{\alpha}, \pi, E, \blacktriangledown) = s'$ **with** $E(x) = [x{\leftarrow}\overline{t}]^{(red,n)}$**.** Then $E = E' : [x{\leftarrow}\overline{t}]^{(red,n)} : E''$ for some environments E', and E''. Remember that terms are considered up to α-equivalence.

     $$\underline{s} = C_{s'}\langle x\downarrow_E\rangle = C_{s'}\langle \overline{t}\downarrow_{E''}\rangle = C_{s'}\langle \overline{t}\downarrow_E\rangle = \underline{s'}$$

     In the chain of equalities we can replace $\overline{t}\downarrow_{E''}$ with $\overline{t}\downarrow_E$ because by well-labeledness the variables bound by E' are fresh with respect to $\overline{t}$.
   - **Case** $s = (F, x, \overline{u} : \pi, E, \blacktriangledown) \rightsquigarrow_{\mathtt{e}_{abs}} (F, \overline{t}^{\alpha}, \overline{u} : \pi, E, \blacktriangledown) = s'$ **with** $E(x) = [x{\leftarrow}\overline{t}]^{abs}$**.** The proof that $\underline{s} = \underline{s'}$ is exactly as in the previous case.
3. *Multiplicative*:
   - **Case** $s = (F, \lambda x.\overline{t}, y : \pi, E, \blacktriangledown) \rightsquigarrow_{\mathtt{m}_1} (F, \overline{t}\{x{\leftarrow}y\}, \pi, E, \blacktriangledown) = s'$**.** Note that $\underline{C_s} = \underline{F}\langle\underline{\pi}\rangle\downarrow_E$ is LO by the decoding invariant (Lemma 6.4). Note also that by the name invariant (Lemma 6.3b) x can only occur in $\overline{t}$. Then:

     $$\begin{array}{rll} \underline{(F, \lambda x.\overline{t}, y : \pi, E, \blacktriangledown)} & = & \underline{F}\langle\langle\lambda x.\overline{t}\rangle\underline{y : \pi}\rangle\downarrow_E \\ & = & \underline{F}\langle\langle(\lambda x.\overline{t})y\rangle\underline{\pi}\rangle\downarrow_E \\ & = & C_{s'}\langle(\lambda x.\overline{t}\downarrow_E)y\downarrow_E\rangle \\ & \to_{\mathtt{LO}\beta} & C_{s'}\langle\overline{t}\downarrow_E\{x{\leftarrow}y\downarrow_E\}\rangle \\ & =_{L.6.3b\&L.3.2} & C_{s'}\langle\overline{t}\{x{\leftarrow}y\}\downarrow_E\rangle \\ & = & \underline{(F, \overline{t}\{x{\leftarrow}y\}, \pi, E, \blacktriangledown)} \end{array}$$

   - **Case** $s = (F, \lambda x.\overline{t}, \overline{u} : \pi, E, \blacktriangledown) \rightsquigarrow_{\mathtt{m}_2} (F, \overline{t}, \pi, [x{\leftarrow}\overline{u}]^l : E, \blacktriangledown) = s'$ **with** $\overline{u}$ **not a variable.** Note that $\underline{C_{s'}} = \underline{F}\langle\langle\cdot\rangle\underline{\pi}\rangle\downarrow_E = \underline{F\downarrow_E}\langle\langle\cdot\rangle\underline{\pi\downarrow_E}\rangle$ is LO by the decoding invariant (Lemma 6.4). Note also that by the name invariant

(Lemma 6.3b) x can only occur in $\bar{t}$. Then:

$$\begin{array}{lll}
\underline{(F, \lambda x.\bar{t}, \overline{u} : \pi, E, \blacktriangledown)} & = & \underline{F}\langle\langle\lambda x.\bar{t}\rangle\underline{\overline{u} : \pi}\rangle\downarrow_E \\
& = & \underline{F}\langle\langle(\lambda x.\bar{t})\overline{u}\rangle\underline{\pi}\rangle\downarrow_E \\
& = & \underline{F\downarrow_E}\langle\langle(\lambda x.\bar{t}\downarrow_E)\overline{u}\downarrow_E\rangle\underline{\pi\downarrow_E}\rangle \\
& \to_{\mathtt{LO}\beta} & \underline{F\downarrow_E}\langle\langle\bar{t}\downarrow_E\{x{\leftarrow}\overline{u}\downarrow_E\}\rangle\underline{\pi\downarrow_E}\rangle \\
& =_{L.6.3b\&L.3.2} & \underline{F\downarrow_E}\langle\langle\bar{t}\{x{\leftarrow}\overline{u}\}\downarrow_E\rangle\underline{\pi\downarrow_E}\rangle \\
& = & \underline{F}\langle\langle\bar{t}\{x{\leftarrow}\overline{u}\}\rangle\underline{\pi}\rangle\downarrow_E \\
& =_{L.6.3b\&L.3.1} & \underline{F}\langle\langle\bar{t}\{\{x{\leftarrow}\overline{u}\}\}\rangle\underline{\pi}\rangle\downarrow_E \\
& =_{L.6.3b} & \underline{F}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\{\{x{\leftarrow}\overline{u}\}\}\downarrow_E \\
& = & \underline{F}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\downarrow_{[x\leftarrow\overline{u}]^l : E} \\
& = & \underline{(F, \bar{t}, \pi, [x{\leftarrow}\overline{u}]^l : E, \blacktriangledown)}\square
\end{array}$$

Proof of the Progress Lemma (Lemma 8, p. 13)

A simple inspection of the machine transitions shows that final states have the form $(\epsilon, \bar{t}, \epsilon, E, \blacktriangle)$. Then by the normal form invariant (Lemma 6.2a) $\underline{s} = \bar{t}\downarrow_E$ is β-normal. $\square$

Proof of the Weak Bisimulation Theorem (Thm 2, p. 13)

1. By induction on the length $|\rho|$ of ρ, using the one-step weak simulation lemma (Lemma 7). If ρ is empty then the empty derivation satisfies the statement. If ρ is given by $\sigma : s \rightsquigarrow^* s''$ followed by $s'' \rightsquigarrow s'$ then by *i.h.* there exists $e : \underline{s} \to^*_{\mathtt{LO}\beta} \underline{s''}$ s.t. $|e| = |\sigma|_{\mathtt{m}}$. Cases of $s'' \rightsquigarrow s'$:
   (a) *Commutative or Exponential*. Then $\underline{s''} = \underline{s'}$ by Lemma 7.1 and Lemma 7.2, and the statement holds taking $d := e$ because $|d| = |e| =_{i.h.} |\sigma|_{\mathtt{m}} = |\rho|_{\mathtt{m}}$.
   (b) *Multiplicative*. Then $\underline{s''} \to_{\mathtt{LO}\beta} \underline{s'}$ by Lemma 7.3 and defining d as e followed by such a step we obtain $|d| = |e| + 1 =_{i.h.} |\sigma|_{\mathtt{m}} + 1 = |\rho|_{\mathtt{m}}$.
2. We use $\mathtt{nf}_{\mathtt{ec}}(s)$ to denote the normal form of s with respect to exponential and commutative transitions, that exists and is unique because $\rightsquigarrow_{\mathtt{c}} \cup \rightsquigarrow_{\mathtt{e}}$ terminates (termination is given by forthcoming Theorem 3 and Theorem 4, that are postponed because they actually give precise complexity bounds, not just termination) and the machine is deterministic (as it can be seen by an easy inspection of the transitions). The proof is by induction on the length of d. If d is empty then the empty execution satisfies the statement. If d is given by $e : \bar{t} \to^*_{\mathtt{LO}\beta} w$ followed by $w \to_{\mathtt{LO}\beta} u$ then by *i.h.* there is an execution $\sigma : s \rightsquigarrow^* s''$ s.t. $w = \underline{s''}$ and $|\sigma|_{\mathtt{m}} = |e|$. Note that since exponential and commutative transitions are mapped on equalities, σ can be extended as $\sigma' : s \rightsquigarrow^* s'' \rightsquigarrow^*_{\mathtt{e}_{red}, \mathtt{e}_{abs}, \mathtt{c}_{1,2,3,4,5,6}} \mathtt{nf}_{\mathtt{ec}}(s'')$ with $\underline{\mathtt{nf}_{\mathtt{ec}}(s'')} = w$ and $|\sigma'|_{\mathtt{m}} = |e|$. By the progress property (Lemma 8) $\mathtt{nf}_{\mathtt{ec}}(s'')$ cannot be a final state, otherwise $w = \underline{\mathtt{nf}_{\mathtt{ec}}(s'')}$ could not reduce. Then $\mathtt{nf}_{\mathtt{ec}}(s'') \rightsquigarrow_{\mathtt{m}} s'$ (the transition is necessarily multiplicative because $\mathtt{nf}_{\mathtt{ec}}(s'')$ is normal with respect to the other transitions). By the one-step weak simulation lemma (Lemma 7.3) $\underline{\mathtt{nf}_{\mathtt{ec}}(s'')} = w \to_{\mathtt{LO}\beta} \underline{s'}$ and by determinism of $\to_{\mathtt{LO}\beta}$ (Lemma 1)

$\underline{s'} = u$. Then the execution ρ defined as σ' followed by $\mathtt{nf}_{\mathtt{ec}}(s'') \rightsquigarrow_{\mathtt{m}} s'$ satisfy the statement, as $|\rho|_{\mathtt{m}} = |\sigma'|_{\mathtt{m}} + 1 = |\sigma|_{\mathtt{m}} + 1 = |e| + 1 = |d|$. □

Proof of the Exponentials vs Multiplicatives Theorem (Thm 3, p. 15)

1. We prove that $|\sigma|_{\mathtt{e}} \leq |E|$. The statement follows from the environment size invariant (Lemma 9.2), for which $|E| \leq |\rho|_{\mathtt{m}}$.
   If $|\sigma|_{\mathtt{e}} = 0$ it is immediate. Then assume $|\sigma|_{\mathtt{e}} > 0$, so that there is a first exponential transition in σ, *i.e.* σ has a prefix $s' \rightsquigarrow_{\mathtt{c}}^* \rightsquigarrow_{\mathtt{e}} s'''$ followed by an execution $\tau : s''' \rightsquigarrow^* s''$ such that $|\tau|_{\mathtt{m}} = 0$. Cases of the first exponential transition $\rightsquigarrow_{\mathtt{e}}$:
   - Case $\rightsquigarrow_{\mathtt{e}_{abs}}$: the next transition is necessarily multiplicative, and so τ is empty. Then $|\sigma|_{\mathtt{e}} = 1$. Since the environment is non-empty (otherwise $\rightsquigarrow_{\mathtt{e}_{abs}}$ could not apply), $|\sigma|_{\mathtt{e}} \leq |E|$ holds.
   - Case $\rightsquigarrow_{\mathtt{e}_{(red,n)}}$: we prove by induction on n that $|\sigma|_{\mathtt{e}} \leq n$, that gives what we want because $n \leq |E|$ by Remark 1. Cases:
     - $n = 1$) Then τ has the form $s''' \rightsquigarrow_{\mathtt{c}}^* s''$ by Lemma 11.1, and so $|\sigma|_{\mathtt{e}} = 1$.
     - $n = 2$) Then τ is a prefix of $\rightsquigarrow_{\mathtt{c}}^* \rightsquigarrow_{\mathtt{e}_{abs}}$ or $\rightsquigarrow_{\mathtt{c}}^* \rightsquigarrow_{\mathtt{e}_{(red,1)}}$ by Lemma 11.2. In both cases $|\sigma|_{\mathtt{e}} \leq 2$.
     - $n > 2$) Then by Lemma 11.3 τ is either shorter or equal to $\rightsquigarrow_{\mathtt{c}}^* \rightsquigarrow_{\mathtt{e}_{(red,n-1)}}$, and so $|\sigma|_{\mathtt{e}} \leq 2$, or it is longer than $\rightsquigarrow_{\mathtt{c}}^* \rightsquigarrow_{\mathtt{e}_{(red,n-1)}}$, *i.e.* it writes as $\rightsquigarrow_{\mathtt{c}}^*$ followed by an execution τ' starting with $\rightsquigarrow_{\mathtt{e}_{(red,n-1)}}$. By *i.h.* $|\tau'| \leq n - 1$ and so $|\sigma| \leq n$.
2. This is a standard reasoning: since by local boundedness (the previous point) $\mathtt{m}$-free sequences have a number of $\mathtt{e}$-transitions that are bound by the number of preceding $\mathtt{m}$-transitions, the sum of all $\mathtt{e}$-transitions is bound by the square of $\mathtt{m}$-transitions. It is analogous to the proof of Theorem 7.2.3 in [6].

□

Proof of Commutatives vs Exponentials Theorem (Thm 4, p. 15)

1. We prove a slightly stronger statement, namely $|\rho|_{\blacktriangledown\mathtt{c}} + |\rho|_{\mathtt{m}} \leq (1 + |\rho|_{\mathtt{e}}) \cdot |t|$, by means of the following notion of size for stacks/frames/states:

$$\begin{array}{rclrcl} |\epsilon| &:=& 0 & |x : F| &:=& |F| \\ |\bar{t} : \pi| &:=& |\bar{t}| + |\pi| & |\bar{t}\Diamond\pi : F| &:=& |\pi| + |F| \\ |(F, \bar{t}, \pi, E, \blacktriangledown)| &:=& |F| + |\pi| + |\bar{t}| & |(F, \bar{t}, \pi, E, \blacktriangle)| &:=& |F| + |\pi| \end{array}$$

By direct inspection of the rules of the machine it can be checked that:
- *Exponentials Increase the Size*: if $s \rightsquigarrow_{\mathtt{e}} s'$ is an exponential transition, then $|s'| \leq |s| + |t|$ where $|t|$ is the size of the initial term; this is a consequence of the fact that exponential steps retrieve a piece of code from the environment, which is a subterm of the initial term by Lemma 9.1;
- *Non-Exponential Evaluation Transitions Decrease the Size*: if $s \rightsquigarrow_a s'$ with $a \in \{\mathtt{m}_1, \mathtt{m}_2, \blacktriangledown\mathtt{c}_1, \blacktriangledown\mathtt{c}_2, \blacktriangledown\mathtt{c}_3\}$ then $|s'| < |s|$ (for $\blacktriangledown\mathtt{c}_3$ because the transition switches to backtracking, and thus the size of the code is no longer taken into account);

- *Backtracking Transitions do not Change the Size*: if $s \leadsto_a s'$ with $a \in \{\blacktriangle\mathtt{c}_4, \blacktriangle\mathtt{c}_5, \blacktriangle\mathtt{c}_6\}$ then $|s'| = |s|$.

Then a straightforward induction on $|\rho|$ shows that

$$|s'| \leq |s| + |\rho|_{\mathtt{e}} \cdot |t| - |\rho|_{\blacktriangledown\mathtt{c}} - |\rho|_{\mathtt{m}}$$

i.e. that $|\rho|_{\blacktriangledown\mathtt{c}} + |\rho|_{\mathtt{m}} \leq |s| + |\rho|_{\mathtt{e}} \cdot |t| - |s'|$.
Now note that $|\cdot|$ is always non-negative and that since s is initial we have $|s| = |t|$. We can then conclude with

$$\begin{aligned} |\rho|_{\blacktriangledown\mathtt{c}} + |\rho|_{\mathtt{m}} &\leq |s| + |\rho|_{\mathtt{e}} \cdot |t| - |s'| \\ &\leq |s| + |\rho|_{\mathtt{e}} \cdot |t| \qquad = |t| + |\rho|_{\mathtt{e}} \cdot |t| = (1 + |\rho|_{\mathtt{e}}) \cdot |t| \end{aligned}$$

2. We have to estimate $|\rho|_{\blacktriangle\mathtt{c}} = |\rho|_{\blacktriangle\mathtt{c}_4} + |\rho|_{\blacktriangle\mathtt{c}_5} + |\rho|_{\blacktriangle\mathtt{c}_6}$. Note that
   (a) $|\rho|_{\blacktriangle\mathtt{c}_4} \leq |\rho|_{\blacktriangledown\mathtt{c}_2}$, as $\leadsto_{\blacktriangle\mathtt{c}_4}$ pops variables from F, pushed only by $\leadsto_{\blacktriangledown\mathtt{c}_2}$;
   (b) $|\rho|_{\blacktriangle\mathtt{c}_5} \leq |\rho|_{\blacktriangle\mathtt{c}_6}$, as $\leadsto_{\blacktriangle\mathtt{c}_5}$ pops pairs $\bar{t}\Diamond\pi$ from F, pushed only by $\leadsto_{\blacktriangle\mathtt{c}_6}$;
   (c) $|\rho|_{\blacktriangle\mathtt{c}_6} \leq |\rho|_{\blacktriangledown\mathtt{c}_3}$, as $\leadsto_{\blacktriangle\mathtt{c}_6}$ ends backtracking phases, started only by $\leadsto_{\blacktriangledown\mathtt{c}_3}$.
   Then $|\rho|_{\blacktriangle\mathtt{c}} \leq |\rho|_{\blacktriangledown\mathtt{c}_2} + 2|\rho|_{\blacktriangledown\mathtt{c}_3} \leq 2|\rho|_{\blacktriangledown\mathtt{c}}$.
3. We have $|\rho|_c = |\rho|_{\blacktriangledown\mathtt{c}} + |\rho|_{\blacktriangle\mathtt{c}} \leq_{P.2} |\rho|_{\blacktriangledown\mathtt{c}} + 2|\rho|_{\blacktriangledown\mathtt{c}} \leq_{P.1} 3 \cdot (1 + |\rho|_{\mathtt{e}}) \cdot |t|$. □

Proof of the Useful MAM Overhead Bound Theorem (Thm 5, p. 15)

1. By definition, the length of the execution ρ simulating d is given by $|\rho| = |\rho|_{\mathtt{m}} + |\rho|_{\mathtt{e}} + |\rho|_c$. Now, by Theorem 3.2 we have $|\rho|_{\mathtt{e}} = O(|\rho|_{\mathtt{m}}^2)$ and by Theorem 4.3 we have $|\rho|_c = O((1 + |\rho|_{\mathtt{e}}) \cdot |t|) = O((1 + |\rho|_{\mathtt{m}}^2) \cdot |t|)$. Therefore, $|\rho| = O((1 + |\rho|_{\mathtt{e}}) \cdot |t|) = O((1 + |\rho|_{\mathtt{m}}^2) \cdot |t|)$. By Theorem 2.2 $|\rho|_{\mathtt{m}} = |d|$, and so $|\rho| = O((1 + |d|^2) \cdot |t|)$.
2. The cost of implementing ρ is the sum of the costs of implementing the multiplicative, exponential, and commutative transitions. Remember that the idea is that variables are implemented as references, so that environment can be accessed in constant time (*i.e.* they do not need to be accessed sequentially):
   (a) *Commutative*: every commutative transition evidently takes constant time. At the previous point we bounded their number with $O((1 + |d|^2) \cdot |t|)$, which is then also the cost of all the commutative transitions together.
   (b) *Multiplicative*: a $\leadsto_{\mathtt{m}_1}$ transition costs $O(|t|)$ because it requires to rename the current code, whose size is bound by the size of the initial term by the subterm invariant (Lemma 9.1a). A $\leadsto_{\mathtt{m}_2}$ transition also costs $O(|t|)$ because executing the Checking AM on $\overline{u}$ takes $O(|\overline{u}|)$ commutative steps (Theorem 1.2), commutative steps take constant time, and the size of $\overline{u}$ is bound by $|t|$ by the subterm invariant (Lemma 9.1b). Therefore, all together the multiplicative transitions cost $O(|d| \cdot |t|)$.
   (c) *Exponential*: At the previous point we bounded their number with $|\rho|_{\mathtt{e}} = O(|d|^2)$. Each exponential step copies a term from the environment, that by the subterm invariant (Lemma 9.1d) costs at most $O(|t|)$, and so their full cost is $O((1 + |d|) \cdot |t|^2)$ (note that this is exactly the cost of the commutative transitions, but it is obtained in a different way).

   Then implementing ρ on RAM takes $O((1 + |d|) \cdot |t|^2)$ steps. □